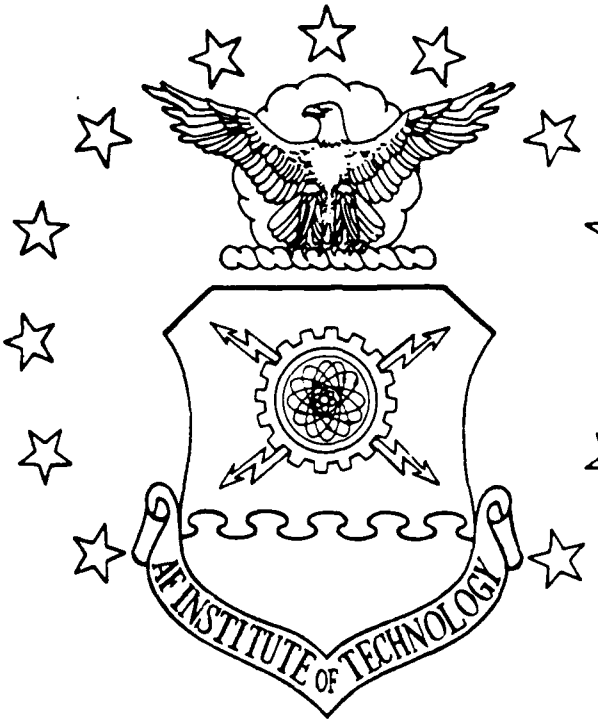

DESIGN AND IMPLEMENTATION OF THE NESTED RELATIONAL DATA MODEL UNDER THE EXODUS EXTENSIBLE DATABASE SYSTEM

THESIS

Michael Anthony Mankus
Captain, USAF

AFIT/GCS/ENG/89D-11

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/89D-11

DESIGN AND IMPLEMENTATION OF THE NESTED RELATIONAL DATA MODEL UNDER THE EXODUS EXTENSIBLE DATABASE SYSTEM

THESIS

Michael Anthony Markus
Captain, USAF

AFIT/GCS/ENG/89D-11

AFIT/GCS/ENG/89D-11

DESIGN AND IMPLEMENTATION OF THE NESTED RELATIONAL DATA MODEL UNDER THE EXODUS EXTENSIBLE DATABASE SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Michael Anthony Mankus, B.S.E.E.

Captain, USAF

December, 1989

Acknowledgments

I am indebted to a number of people who helped and supported me throughout the thesis project. First, my advisor, Major Mark Roth, who did his best in keeping me on track throughout the thesis. Thanks also go out to Captain James Kirkpatrick who helped me in learning the C programming language and all the peculiarities of the UNIX system. I would also like to thank my mother and father who have always inspired me in all my endeavors. Finally, I would like to thank my lovely wife Lynne whose patience and understanding reminded me to never forget the important things in life.

Michael Anthony Markus

Table of Contents

List of Figures

Figure — Page

Figure Page

Abstract

The problem addressed in this thesis effort concerns the design and implementation of the nested relational data model. The data model is designed within the Exodus extensible architecture. Although a large amount of theory exists with the model, no vehicle has been available to implement the concepts. The objective of the model is to increase performance of non-traditional databases by modeling real-world objects in the problem domain into nested relations within the software domain of Exodus.

Exodus is used to implement several components essential to the data model. First, the concept of nested relations is realized, and then a parser is developed to create and maintain a data dictionary. The Colby relational algebra is used to form the query tree for the query optimizer, and a plan tree permits the code to be generated for the query. Operator methods are developed for the query to be subsequently executed.

The nested relational data model was implemented using the Exodus architecture. The query tree was built and the code generated for the architecture's compiler. Operator methods were implemented for the project, select, and natural join operators. Because the data model can be implemented, more non-traditional databases can be developed with efficient components.

Design and Implementation of the Nested Relational Data Model Under The Exodus Extensible Database System

I. Introduction

1.1 Background

1.1.1 Historical Perspective. A large segment of the population is becoming increasingly aware of just how much data and information processing continues to revolutionize everyday living. The complex technologies of today and tomorrow continue to advance the means of manipulating electonic data in its many forms to achieve some end. For example, an individual can now shop for personal goods and services with the help of a home computer, while businesses can use past sales data to develop future strategies. In either case, large amounts of data are involved and require effective and efficient processing with the efforts of some type of database system.

A majority of existing database systems provide the features to retrieve, modify, and manipulate data within the well-defined rules of the relational data model (8). They revolve around the concept of files and records, vital entities in today's fast-paced business transaction environment. Because files and records map extremely well into the tabular format of the relational model, software developers have aggressively pursued the design of complex, integrated database systems to accomodate this large, expanding market.

The relation is regarded as a table of records with common properties or attributes. These common attributes are the fields defined on a record within the relation. Several operations may be performed on relations to extract or modify the data within the database. The operations are firmly based in the relational algebra and calculus, and these mathematical concepts continue to provide the foundation for further work in relational database models.

The wide acceptance of the relational model provides the framework for most successful database systems and an enterprise designing a database for its particular needs

must adhere to the model and its prescribed operations. Although not a deterrent for most enterprises that automate record-type accounting procedures, the model is not meeting the needs of potential users seeking non-traditional database support.

1.1.2 Nontraditional Database Applications. A growing need for database systems is spreading from the traditional applications, as mentioned above, to what are commonly known as non-traditional applications. Many research and engineering enterprises, with their propensity to collect and manipulate data, are recognizing the need for database management systems. However, many are forced to design ad hoc or customized database systems for their particular applications, and the design normally revolves around some flat file structure with limited operations. In essence, the relational model and its operations are not providing an adequate vehicle upon which non-traditional applications can be built. Computer-aided design/computer-aided manufacturing/computer-aided engineering (CAD/CAM/CAE), forms management, and imagery/voice data are just a few examples of non-traditional database applications.

1.1.3 Extending the Relational Model. To accomodate this growing area of database system design, attempts have been underway over the past decade to expand the capabilities of the relational model. With the growing popularity of object-oriented programming, several commercial products are designing abstract data typing (ADT) facilities in an attempt to design database systems for these non-traditional applications. By allowing the formulation of ADTs, database designers are adding new operations for these non-traditional database concerns, to more readily model the real world. In addition, object-oriented databases are introducing the notions of inheritance and message-passing to enhance the object-oriented environment (3).

1.1.4 Separating the Architecture from the Data Model. This has not gone unnoticed by database designers. The advent of new data models beyond the context of the relational model has forced a re-evaluation of database design. Several systems, most of an object-oriented flavor, have departed from the notion of one model and one architecture. These new design packages take a toolkit approach toward the design of database systems.

They provide the basic necessities that a database system requires: the storage system and the programming facilities to develop the system software. Several of these current current systems under development include POSTGRES, GEMSTONE, GENESIS, and EXODUS (4).

1.1.5 The Nested Relational Model. The nested relational model is an extension of the traditional relational model. Besides retaining the traditional relational operations, most research efforts point toward the addition of two more operators: nest and unnest. With the augmentation of these operators, the nesting of relations within relations is theoretically possible. That is, the attributes defined on some relation may actually be set-valued or relation-valued domains. The nested relational data model appears to be a possible alternative to support non-traditional database system applications and the architectures mentioned above provide the basic means to design and implement such a system.

1.2 Purpose of Thesis

This thesis will use the EXODUS architecture to implement the nested relational data model. EXODUS, an acronym for EXtensible Object-oriented Database System, is the property of the University of Wisconsin and is an ongoing research project within the university's computer science department. The extensibility of EXODUS lies in its provision of the necessary tools and components to develop application-specific database systems. The system compiler, used for the compilation of system software, is based on the E programming language, an extension of the object-oriented programming language C++.

The present direction of database design actually strips away many of the components and facilities commonly available with today's systems. In this manner, standardized tools and components may be implemented on top of a particular architecture such as EXODUS. If generic or all-purpose components are already available in the system's libraries, the database engineer, or DBE, adds the necessary ones in a modular fashion to avoid "re-inventing the wheel."

1.2.1 The Problem. The main objective of this thesis effort is to construct a working nested relational database management system utilizing the tools provided by the Exodus architecture. The overall problem, then, is to accurately reflect the model's features within the constraints imposed by Exodus. A breakdown of the problem entails examining each one of the system components and tailoring it to the specifics of the nested relational data model. Component considerations range from choosing a suitable relational algebra to designing and implementing operator methods for actual data extraction. Six primary areas of concern to achieve the thesis objective are discussed below.

Relational algebras for the nested relational data model are primarily extensions of the traditional relational data model as first presented in (6). In this research, two of the extended algebras under consideration include those found in (7) and (13). The Colby algebra was chosen because it has an inherent and pleasing approach in its interaction with nested relations, and may prove less difficult to implement under an object-oriented development environment. The basic problem, then, involves accurately transforming the Colby relational algebra into the software domain defined by Exodus.

The catalog manager is an important, necessary component for any type of database system. It is here where information is maintained for all databases and their relations associated with the system. The catalog manager permits various system components to access the data dictionary and obtain information crucial to their own operation. The data dictionary consolidates this information into data tables defining different aspects of the database system as a whole. A system using the nested relational data model must account for the nested attributes and in which relations they currently exist. The design problems with respect to the catalog manager and data dictionary center on these tables and the routines which access them. The design must consider the number of tables which must be created for providing minimum system functionality as well as the attributes which define the properties of the tables. The routines must allow all components requiring catalog support uniform access into the data dictionary.

The third area of concern focuses on the structure of query tree. The tree reflects the nature of the system's relational algebra, so the tree structure design and the relational algebra chosen coincide very closely with each other. Since Colby's algebra is to be im-

plemented, the operator specifications must be transformed into a structure recognized by the optimizer. All query data entered by way of the parser must be placed into the space allocated by the nodes comprising the query tree, each node representing an operator of the relational algebra. Because of the recursive nature of the algebra, additional auxiliary nodes maintain the condition and navigation information required for descending into the nested structure of relations. These auxiliary nodes represent the operator lists for each of the respective relational operators. The problem highlights the need to ensure the relational algebra and the tree's data structure match the intended nature of the algebra.

To enable the proper operation of the catalog manager with respect to the data dictionary, and to ensure that the relational algebra is properly translated into a query tree, another component must be designed to tie these entities together into the system. The parser provides the mechanisms to allow this type of system functionality to occur. Since Exodus does not provide any type of parser design capabilities, the UNIX tool known as YACC (for Yet Another Compiler Compiler) will be used for this purpose. The difficulty lies in establishing the proper grammar rules for system operation as a whole.

The fifth major area of concern involves the generation of the E source code to execute the query. Although this is situated after the query optimizer, the optimizer will not be of concern in this project. Because of its expected complexity, a doctoral student is developing the optimizer component. Upon exiting the query optimizer, the query tree has become a plan tree, structured in such a way to efficiently access the database with respect to the current query. Because the data structure is unique to the Colby algebra, routines must be developed to accurately walk the plan and its auxiliary list and condition structures, inspecting the information and generating the required source code. Once all of the source code is generated, the routines dynamically link its object code to the implemented operator methods. The problem, therefore, is twofold. First, accurately generate the E code and, second, link it with the required operator methods to produce the executable module.

The final problem focuses on the structure of the relation, which must logically and physically allow for the nesting of relations. Procedural programming languages have not traditionally permitted such structures, and DBEs have had to resort to ad hoc measures

to achieve this end. However, since E is a database system language, mechanisms added to C++ appear to bridge the gap between the logical and physical nature of nested relations. This effort will take advantage of these features to design and implement nested structures.

1.2.2 Scope of the Thesis. By indicating the six problem areas of designing and implementing the nested relational data model within the Exodus framework, the scope of this thesis effort was narrowed to achieve this objective. The data dictionary contains enough information to allow proper operation of the database system, while the catalog manager has the routines needed to access the data tables. The routines are able to be invoked by any other component in the system. Project, select, and natural join are the operators implemented within the database. The parser is able to parse the query into the proper tree structure as well as allow for the creation of nested relations by the user. The walking routines are able to walk the plan tree and generate the proper E code, dynamically linking the query to the operator methods. The scope outlined above provides a minimum functionality for the nested relational database system.

1.3 Sequence of Presentation

First, the nested relational data model is examined in Chapter II, followed by a discussion of the Exodus Extensible Architecture in Chapter III. After presenting this background information, the design and implementation of the data model under Exodus is demonstrated in Chapter IV. Finally, conclusions are provided on this effort, with recommendations for future research.

II. The Nested Relational Data Model

2.1 Introduction

The relational data model is the primary vehicle for most of today's sophisticated database management systems. The model is defined within sound mathematical principles, and it is this very foundation which permits accurate manipulation of relations and their corresponding data members. By understanding and applying the relational functions to these abstract structures, an appreciation of the model's strengths demonstrates an increasing role for further extension. New data models are a result of these extensions, including the nested relational data model.

Since the relational data model lends credence to the nested relational data model, features unique to this traditional model must initially be examined. A constructive approach of this type attempts to eliminate misconceptions of the nested relational data model. Once an appropriate amount of background is presented for the traditional relational data model, the properties of the extended version of this model will be discussed.

2.2 The Relational Data Model

The structure of the relational data model, sometimes prefixed by the "conventional" or "traditional" adjectives, may be visualized in two different forms. First, the logical nature of the data model is the most recognizeable structure in the object-oriented arena, indicating some particular entity with its various properties. The abstractness of the model allows the mapping of traits of real-world objects into the software domain, where the objects may then be invoked to behave as if in the real-world. Second, the physical nature of the data model attempts to separate these traits into a straightforward, tabular format. Instead of abstractly visualizing an object with specific properties unique to the object, the physical table or relation permits actual object data to be compared to other objects with similar properties.

A table of values is the most likely representation of an object modeled under the relational data model. Rows and columns are essential elements of a relation, where each column indicates a particular attribute of an object, while each row represents an entire

instance of an object defined by the relation. Attributes must rely on already defined types such as the integer, float, or character types available in most software environments. Of course these types may represent other types or instances in a higher abstract sense. For instance, the integer '1' is commonly a stand-in for the Boolean variable 'TRUE', while '0' reverts to the Boolean 'FALSE'. In this case, integers are used for the Boolean type.

An example of a relation should help emphasize the logical and physical nature it attempts to model. From a logical standpoint, the relation represents some object in the real-world domain. An often used object is *person* since its properties may easily be understood and subsequently derived. To narrow down the properties attributable to any one person, the actual *person* object may be scaled down to a *child* object, where the *child* object most likely has ties to some other object such as an *employee* object. That is, the relation defines the attributes of children belonging to employees of some particular corporation.

To relate each of the *child* object instances to *employee* objects presumably residing elsewhere in the system, one *child* attribute must be able to directly link its object instances to those *employee* instances having children. For this purpose, the *emp_ssn* attribute, representing an employee's *social security number*, becomes one of the key attributes which provides uniqueness to the *child* objects. Other attributes of a *child* include *name*, *age*, and *sex*. From the countless possibilities which may define a *child*, the database requires only a few attributes central to the application for which the database was developed.

From a physical standpoint the relation consolidates the attributes of the object into a coherent structure. Columns of the table or relation, shown in Figure 2.1, represents each of the *child* attributes, while each of the rows or tuples are the specific children instances. In this example, there are ten instances of the *child* object. The attributes, as previously discussed, must be one of the basic types. The *emp_ssn* attribute may be an array of characters or strings, or it can be regarded as an integer with the hyphens inserted at the time of extraction. In the same way, *name* is a string, *age* is an integer, and *sex* is a single character, 'M' or 'F'.

The operators of the relational algebra do not specifically model the behavior of the

emp_ssn	*name*	*age*	*sex*
192-83-7465	Bob	5	M
192-83-7465	Carol	4	F
325-96-0127	Kyle	10	M
325-96-0127	John	12	M
325-96-0127	Lynne	6	F
519-73-3790	Mike	7	M
234-61-9825	Tom	5	M
234-61-9825	Jeremy	4	M
234-61-9825	Tiffany	7	F
234-61-9825	Anele	1	F

Figure 2.1. The *child* relation.

object itself, but rather they permit data to be extracted from the object or modified within it. Although this is a shortcoming for an object-oriented development environment where real-world object properties and behaviors are mapped into the software domain, the relational operators are well-suited for particular applications. Most of these are accounting and transaction-oriented applications which have made the relational data model an indispensible tool in the work place.

2.2.1 The Relational Operators. A number of operations in the relational algebra are possible on relations, including:

- Project
- Select
- Natural Join
- Cartesian Product
- Set Operations

The first three operators are the primary elements necessary for extracting data from relations. The *cartesian product* operator is a generalization of the *natural join*, while the *set operations* include *union*, *intersection*, *difference*, and *divide*. Although the *cartesian product* operator and the *set operations* provide significant functionality to user queries,

the first three permit the necessary extraction of data from relations. Codd (6) and Korth (11) provide a good explanation of each of these operators.

A database populated by relations normally representing objects integral to one another in some fashion, may temporarily force these objects together using the join operator and pulling out instances according to some criteria or obtaining a subset of the properties of the objects. In other words, the three operators are used in various combinations to extract certain data elements from all the existing relations. The conglomeration of objects in the database defines some environment or entity such as a *corporation* which may include items such as a *product* relation and a *resources* relation in addition to the already defined *child* and *employee* relations.

Because many relations within a particular environment are interrelated, a better approach is to combine some of the relations into a single one. This would alleviate using the join operator for two or more relations which are joined for a majority of queries involving them. However, with the voluminous amounts of data that will most likely result from such an endeavor, it is better to keep the relations as separate entities to facilitate more logical query formulation. However, these inter-relationships cannot necessarily be ignored either. Some method or approach must permit a more logical structure to account for these dependencies without altering the basic functionality of the three operators.

2.3 The Nested Relational Data Model

Much of the overall structure and behavior of the relational data model is essential to the underlying framework of the nested relational data model. The relational data model considers its objects as separate entities organized in a horizontal fashion. The join operator links the relations whenever query processing requires a combination of the data. However, a vertical orientation is more appropriate in many of these cases. By permitting only one level of abstraction within a database environment, the real-world environment which a relational database represents is somewhat skewed. A good possibility exists that some of the relations should actually be subordinate to other relations to properly model the real-world objects. This is also known as aggregation. Tables situated horizontally are considered flat relations and are in first normal form (1NF). Their data is in a nondecom-

posable or atomic state. On the other hand, if the relations were to model the real-world, the database would have to become more vertical, placing relations within relations.

Traditional relations place their emphasis on flat tables and the relational operators, while OOD considers objects and the functions that may operate on the objects. The attempt is to model the behavior within the database system. Nested relations, as composite objects (10), require certain data members and functions. An environment must permit the data members within the object to become the atomic and relation-valued attributes, while the functions modeling an object's behavior must be found in its member functions. The relational operators can then also be considered objects, requiring data members to be in the form of nested relations and member functions, when used in unison, to perform the operation. This is the concept behind OOD within the database system.

By inserting relations within relations, this vertical orientation suggests the nesting of relations. According to this scheme, then, a top-level relation represents some particular real-world object, where its composition is based on subobjects. These subobjects may either be basic data types that cannot be broken down any further, or they may be relations which are further comprised of their own particular attributes. In this nested structure, attributes are either nondecomposable or atomic attributes or they are actually relations, commonly referred to as relation-valued attributes. This nested relation structure, as opposed to 1NF relations, is known as non-first normal form ($\neg$1NF).

2.3.1 The Nested Relation. Since nested relations are comprised of relations and atomic attributes, the basic relational structure is still the primary building block. However, the nested relational data model, to form its very structure, must provide a means to insert relations in addition to the basic relational operators. Before discussing the mechanics of these operations, an example is in order.

The nested structure may be illustrated using the two flat relations found in Figure 2.1 and Figure 2.2. To eliminate the joining of the *child* and *employee* relations along their common attribute, *emp_ssn*, before applying the other two operators, the *child* relation can be inserted as a relation-valued attribute of the *employee* relation. In Figure 2.3, the *employee* relation inserts the *child* relation. Now, all other attributes within the *em-*

dept	*emp_name*	*emp_age*	*emp_ssn*
Accounting	Washington, A.B.	33	192-83-7465
Research	Lincoln, C.D.	44	234-61-9825
Development	Kennedy, E.F.	45	519-73-3790
Engineering	Carter, G.H.	38	325-96-0127

Figure 2.2. The *employee* relation.

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*		
				child_name	*child_age*	*sex*
Acct	Washington, A.B.	33	192-83-7465	Bob	5	M
				Carol	4	F
Eng	Carter, G.H.	38	325-96-0127	Kyle	10	M
				John	12	M
				Lynne	6	F
Dev	Kennedy, E.F.	45	519-73-3790	Mike	7	M
Res	Lincoln, C.D.	44	234-61-9825	Tom	5	M
				Jeremy	4	M
				Tiffany	7	F
				Anele	1	F

Figure 2.3. The *employee* nested relation.

ployee relation are atomic, while all attributes within the *children* subrelation are also atomic-valued. In this case, one level of nesting exists within the *employee* relation.

In Figure 2.3, the *emp_ssn* disappears within the *children* attribute since this information exists within the *employee* relation. What once required a join operation and ten tuples, is now condensed into one relation of four tuples. These four tuples are actually supertuples, since the *children* attribute is a set of tuples or subtuples for every tuple of the *employee* relation.

2.3.2 The Relational Algebra. Because the relational structure plays an integral part in the nested relational data model, the relational operations also function on the nested structure, although in an extended form. Although there are several versions of the extended relational algebra to choose from, the Colby relational algebra is used in this thesis. Its apparent simplicity permits easier design and implementation into the query

tree structure. The three major relational operators - project, select, and natural join - are supported, as well as the set operations. Because the nested relational data model requires a method of nesting or unnesting a relation at any level of a nested relation's hierarchy, the nest and unnest operators are added to the algebra.

2.3.2.1 The Nest Operator. Although nested relations can be initially structured at the time of their creation, the ability to nest attributes within an existing relation should be permitted. This entails grouping attributes situated at the same level of nesting and invoking the operator to nest this group of attributes one level deeper.

According to Colby, the nest operator requires three elements. First, an attribute list indicates the group of attributes to be nested to some deeper level. The relation maintaining the said attributes in the list is the second element required for the operator. The third element represents the new relation-valued attribute for which the list of attributes will be nested. As an example, to nest the children attributes shown in Figure 2.4 into the nested version in Figure 2.3, the mathematical formulation is characterized in the following manner:

$$\nu \{ child\_name, child\_age, sex \} (employee) < children >.$$

2.3.2.2 The Unnest Operator. This operator is the dual to the nest operator, although performing the reverse operation. It only requires specifying the relation-valued attribute that is to be unnested. For example, to undo the nesting of Figure 2.3 and obtain the relation in Figure 2.4 once more, the unnest operator, μ, may be used in the following manner:

$$\mu (employee) < \text{children} >.$$

2.3.2.3 The Project Operator. The project operator requires two elements to project out the requested attributes of a nested relation. A project list permits the project operator to reach any attribute, atomic or relation-valued, and project out the contents of a single column or an entire subrelation. The second element is the source relation

dept	*emp_name*	*emp_age*	*emp_ssn*	*child_name*	*child_age*	*sex*
Accounting	Washington, A.B.	33	192-83-7465	Bob	5	M
Accounting	Washington, A.B.	33	192-83-7465	Carol	1	F
Engineering	Carter, G.H.	38	325-96-0127	Kyle	10	M
Engineering	Carter, G.H.	38	325-96-0127	John	12	M
Engineering	Carter, G.H.	38	325-96-0127	Lynne	6	F
Development	Kennedy, E.F.	45	519-73-3790	Mike	7	M
Research	Lincoln, C.D.	44	234-61-9825	Tom	5	M
Research	Lincoln, C.D.	44	234-61-9825	Jeremy	4	M
Research	Lincoln, C.D.	44	234-61-9825	Tiffany	7	F
Research	Lincoln, C.D.	44	234-61-9825	Anele	1	F

Figure 2.4. The unnested version of the *employee* relation.

emp_name	*child_name*
Washington, A.B.	Bob
	Carol
Carter, G.H.	Kyle
	John
	Lynne
Kennedy, E.F.	Mike
Lincoln, C.D.	Tom
	Jeremy
	Tiffany
	Anele

Figure 2.5. Projecting employee and children names.

or a relation resulting from another query. From the nested relation in Figure 2.3, the project operator may be used to project out the *emp_name* and *child_name* attributes in the following manner:

$$\pi \{ \textit{emp\_name}, \textit{children} \{ \textit{child\_name} \} \} (\text{employee}).$$

The ***children*** **attribute** must be specified in the list to allow the operator to descend to the next level of nesting to obtain the data values under the *child_name* attribute. The resulting relation is shown in Figure 2.5.

2.3.2.4 The Select Operator. The select operator requires elements similar to the project operator plus an additional element. Like the project list, the select list

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*		
				child_name	*child_age*	*sex*
Dev	Kennedy, E.F.	45	519-73-3790	Mike	7	M
Res	Lincoln, C.D.	44	234-61-9825	Tom	5	M
				Tiffany	7	F

Figure 2.6. The extended select operator results.

also permits the operator to descend to any depth of the relation. Either a relation or a relation resulting from another query provide the source tuples for the operator. Finally, conditions or predicates must be able to be specified on the top-level atomic attributes or on lower-level relations.

For instance, to select those tuples from the *employee* relation in which an employee's age is greater than forty and the child's age is greater than four will require the following formulation:

$$\sigma\ (employee)\ [emp\_age > 40]\ \{\ children\ [child\_age > 4]\ \}.$$

The top-level condition must immediately follow the relation specified within the parenthesis, while lower-level conditions are found in the select list. The resulting relation may be found in Figure 2.6.

2.3.2.5 The Natural Join. A major difference between the traditional relational algebra natural join and the Colby natural join is the requirement of specifying a join path. This path indicates the attributes on which two relations or relations resulting from queries may be joined. The path is basically the navigation tool to reach the nested attributes for the join.

An example of the natural join involves the *employee* relation in Figure 2.3 and the *school* relation in Figure 2.7. The *school* relation maintains the student data for all those attending the local school district. The formulation of the join is as follows:

$$\bowtie (\text{employee})\ \{\ children\ \}\ (\text{school}).$$

Figure 2.8 is the relation resulting from the join.

school	*level*	*children*		
		child_name	*child_age*	*sex*
North	Mid	John	12	M
		Mary	11	F
		Larry	13	M
South	Elem	Cassy	7	F
		Kyle	10	M
		Lynne	6	F
Green	Elem	Pam	8	F
		Charles	7	M
		Mike	7	M
		Nancy	9	F

Figure 2.7. The *school* relation.

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*			*school*	*level*
				child_name	*child_age*	*sex*		
Eng	Carter	48	325-96-0127	John	12	M	North	Mid
				Kyle	10	M	South	Elem
				Lynne	6	F		
Dev	Kennedy	45	519-73-3790	Mike	7	M	Green	Elem

Figure 2.8. The employee relation joined to the school relation.

2.4 *Summary*

The the nested relational data model obtains most of its structure and functionality from the traditional relational model. The operators in the nested model require extensions to allow them to descend to nested relations. In addition to extending the relational operators, the nested model also uses a nest and unnest operator to allow some restructuring of existing relations.

III. The Exodus Extensible Database Architecture

3.1 Introduction

As an extensible database system, the Exodus architecture is not a database system in and of itself, but rather an integrated package of software tools required for database design and implementation. The tools encompass the development of the primary components necessary to implement a database system for a specific application or a particular data model. For a database system under consideration, a database engineer (DBE) must design and implement a host of system components as well as a number of auxiliary functions to satisfy data model or user requirements. In order to understand the functionality of all required system components, the DBE must be aware of how each of the Exodus software tools will assist in creating or generating the system components for the task at hand.

An analysis of a database system incorporating the nested relational data model exhibits a close resemblance to the structure of traditional relational database systems. To design and implement such a system, a number of the same components are required with similar characteristics. However, an assessment of each of the components reveals an extension of conventional design features to account for the nesting of relations. Under the Exodus environment, the DBE is concerned with the following system components:

- Parser
- Catalog Manager
- Data Dictionary or Schema
- Query Optimizer
- Query Compiler
- E Compiler
- Operator Methods
- Access Methods
- Storage Manager

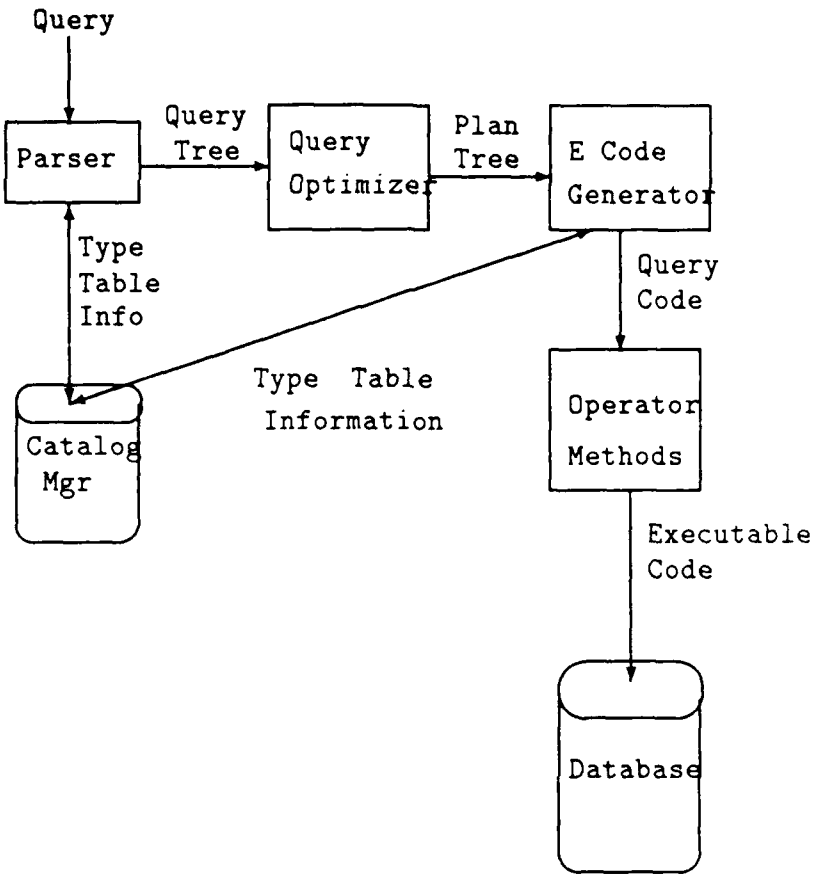

Figure 3.1. A typical Exodus system design.

- Database

A typical system design may be found in Figure 3.1.

An advantage of Exodus focuses on component reuseability, a feature of extensible architectures permitting rapid development of other systems with similar characteristics. The amount of implementation ranges from components produced via input-driven generators to components entirely designed and implemented by the DBE. Properly analyzing system requirements should lend itself to indicating possible areas of component reuse, thereby eliminating duplication of past effort.

The standardization of components for database systems therefore provides some

credence to software engineering principles during system requirements analysis. The Exodus tools provide a frame of reference from which to compose and decompose the intended database system while ascertaining its requirements. The building blocks may be tested according to various test cases to ensure the viability of a particular component under some requirement essential toward the make up of the data model or application. With this in mind, the nested relational data model has unique requirements which must be taken into account at the component level before assembling the entire system. A breakout of each of the Exodus components with respect to the requirements of the nested relational data model provides an outline of the DBE's preliminary design tasks before advancing into detailed design and subsequent implementation.

3.2 The Parser

The main purpose of the parser is to link the user with the database system. Specifically, the user, by way of a data manipulation language and a query language, may access the catalog manager and the query optimizer. These two components are important in their own respect and will be examined in upcoming sections. The parser, by providing the user-interface, translates human-understandable language to one more conducive to database system management.

The user communicates through a standard set of commands, issuing directives which engage the mechanisms of the database. These commands are normally encapsulated into an interface referred to as a query language. A number of query languages have been developed and implemented in traditional relational systems, so it is no wonder that extensions of such languages have been proposed for the nested relational data model. Of the relational query languages, SQL is the most popular to date and extended languages such as SQL/NF for nested relations provide a strong backdrop for implementing the nested model. The DBE must choose an appropriate query language or develop one reflecting the nature of nested relations. Once it is chosen, a high-level parser may be designed and implemented to incorporate the query language.

Exodus does not provide a parser or scanner as a development tool, so an alternative must be found. For this purpose, the popular UNIX tools of YACC (Yet Another

Compiler Compiler) and LEX (Lexical Analyzer) are the most commonly used tools. LEX supports the capability to scan the input for specific commands necessary to indicate to the system what is requested. For instance, if a command to create a new relation was warranted, the words *create relation* will possibly have the associated tokens of T_CREATE T_RELATION, signifying that a create and relation token are to be sent to the parser. Since numeric and character data are essential for database operation, generic scanner formulas for floats, integers, and strings must be developed. However, since scanners with this type of data have been developed before, their formulation will closely match earlier representations found in other scanners. Other types of input required for scanning include the equality and inequality operators, the set operators, data dictionary commands, and query commands. Tokens will have to be returned for each one of these commands or inputs. Regarded as a single component, the scanner is the first subcomponent of the interface with the database user, and tokens from user commands are sent to the parser subcomponent for assembling the user's wishes into database commands. In other words, the scanner and parser subcomponents comprise the "parser" component, since the parser subcomponent cannot operate without a scanning operation.

Once a token is sent from the lexical analyzer, the parser attempts to match one of the grammar rules containing the tokens within the parser. YACC uses a left to right resolution method to determine what the user's command entails. Within each of the grammar rules, actions may be taken to direct the database system to carry out a specific operation.

To illustrate this in more detail, an examination from the earlier *create relation* example is in order. Once the T_CREATE and T_RELATION tokens have been sent to the parser, the parser will most likely have some action to perform. In this case, the actions invoke procedures to allocate and input a relation. Once the operations are completed, the control returns to the user by way of the parser.

3.3 The Catalog Manager

The catalog manager is the mechanism responsible for ensuring proper organization of relations within the database. Data tables which identify and track previously created

relations and their attributes are overseen by this component. An essential task includes routing relation and attribute identifiers to their correct position within the data tables. Whenever a new database is created or a current one opened, all references to any of the relations and their associated attributes are handled by the mechanisms implemented by the catalog manager. The catalog manager's tables must be linked in some manner to retain the correspondence among the various identifiers and permit an efficient and accurate traversal of all tables for search of fundamental system data. A design decision may lie in how to link the tables together. Methods such as using pointers or indexes to match table columns together are simple and establish a clean connection between tables.

In comparison to traditional relational database system catalog managers, many similarities exist for catalog management of nested relations. For a nested relational database system, the catalog manager has to ensure all relations and nested relations are handled reliably and without confusion as to what is a relation and what is a relation-valued attribute embedded within a relation. That is, nested relations cannot be confused with top-level relations or schemas. In addition, duplicate atomic or relation-valued attribute identifiers must always be clearly differentiated to eliminate any confusion which may arise among the various relations in the data dictionary.

The catalog manager may be designed in various ways, but it should take advantage of the resident data model's own structure. For instance, relational database systems normally use relations as the catalog manager's vehicle for maintaining user-defined relations within the database system itself(1). With this perspective, the nested relational database system should take advantage of nesting certain attributes of its catalog tables to manage database tables defined by users of the system. Catalog manager tables can be designed to maintain information of relations within the database, maintain all the attributes of every relation within the database, and maintain those relation-valued attributes identifying nested relations within the database. Information required by the catalog manager to load its tables will be obtained through the scanning and parsing of relation creation commands. Once this information is passed from the scanner, through the parser, and to the catalog manager, the data must be input into its tables in a particular format defined within the system and commonly referred to as the data dictionary.

3.4 The Data Dictionary

The data dictionary is a component operating in conjunction with the catalog manager to support the organization of symbol identifiers and statistical information necessary to maintain relations within the database system. Symbol identifiers include all unique entities defined by database users which may describe individual relations or parts of a relation. Information associated with each identifier, characterizes the state of the particular entity and its placement with respect to all other symbols within the data dictionary.

There are two primary types of data available within the data dictionary. First, atomic data will normally consist of one of the three basic data types found in most programming languages: integer, float, or character. Second, a database must distinguish the different table or relation types available for relation creation (12). A table type defines the structure of relation that may be used for future instantiation as relation or a relation-valued attribute. That is, it provides the relation definition which is to be nested within another relation. A table type is considered to be defined *on the fly* because it is a new entry into the database and uses other data types for its attributes.

The symbol table is comprised of the table types defined for the nested relational database system along with their individual atomic or relation-valued attributes. Once a relation is created, using of the existing table types, the catalog manager parses the relation information and routes it to the relation table rather than the symbol table.

As expressed earlier, the data dictionary is a repository of symbol identifiers and associated information characterizing each of the symbols in the symbol table. The same requirements hold true for the data dictionary's relation table. Several of these elements in combination with each other provide the uniqueness each symbol or relation requires for the catalog manager to distinguish among the considerable amount of entries possible within the data dictionary. The design of any type of database system will require a minimum set of table characteristics to enable the catalog manager to easily access the necessary symbol or relation and acquire the information requested by another system component. Under the nested relational database structure, the attributes required for these two tables normally include the index, name, data type, and any reference to another entity in the

same table or the other table.

3.4.1 Name. One property to which the user is most likely to relate is the name of the table entry. Upon entry of a table type or relation, the user's chosen name is added to the catalog manager tables.

The use of a name does permit useful dissemination of data dictionary information, although it may be used more than once among the attributes of the defined table types in the symbol table. Other attributes are acquired for each of the table entries to eliminate any ambiguity during catalog manager scans of the table data.

3.4.2 Type. A system designed under an object-oriented environment emphasizes strong typing facilities. Data typing lends itself best to mapping real-world problems and their solution domains into the corresponding software domains. The symbol table ensures all data types are preserved as long as the object is defined in the data dictionary.

3.4.3 Level. The level of a symbol in the symbol table provides further information about a specific object in the database system. There are two basic levels available for an object within the nested relational data model. Although a relation may have any number of nesting levels, the only distinction among the elements within the symbol table focuses upon the relationhip between a table type and its attributes, whether atomic or relation-valued. Therefore, the inherent number of levels is two. That is, a symbol is at the *table* or the *attribute* level. The *level* property attempts to augment the *attribute* property by supporting the distinction between table symbols and relation-valued symbols. Level is of no concern within the relation table.

3.4.4 Number. This data dictionary attribute capsulizes the definition of a table type through application of a quantitative measure. After parsing of the table type definition, the number of attributes, atomic and relation-valued, provide a gauge as to the size of the relation. Use of a table's size number allows succinct scans of relations and their nested relations. In addition to using the number as a size metric for table types, the size of atomic attributes defined as character arrays may also be stored here.

3.4.5 Parent. To further enhance the cohesiveness of the data dictionary, each of the items in the symbol table bears a direct reference to its defining table type. In conjunction with the *level* designation, the *parent* attribute identifies a particular symbol's table type. Symbols at the table level use an arbitrary level designator since there is no defining table type or database at this point. If more than one database is allowed to exist within the system, then each table level attribute may enter the name of the database in which it resides.

A number of other statistics may increase the effectiveness of the data dictionary, but the above attributes apparently provide adequate coverage of necessary data elements within the database. The number of attributes per table entry is small enough not to overburden the database system, while providing a fully functional nested relational database system.

3.5 Query Optimizer

The query optimizer is a standard system component capable of transforming a user's input query tree into an efficient query access plan. In effect, the component optimizes the information embedded within the query in accordance with rules devised by the DBE for a nested relational database system. The query tree is formed according to DBE data structures during parsing of the input query. As the tree progresses through the optimizer, information within the tree nodes are used for optimization purposes, and the resulting plan is deposited at the output of the component.

The query tree is built node-by-node based upon information parsed from the user's database request. The tree is linked together by means of query nodes, each node representing the objective of a relational algebra operator.

Because the nested relational data model relies upon an extended relational algebra as well as the nest and unnest operators, each node must earmark enough space for each of the operators and their respective list structures. Depending upon the specific operator, lists will be necessary data structures to hold auxiliary data permitting the operators to descend relations and their relation-valued attributes under the current query's considera-

tion. The query tree is a binary structure comprised of query or tree nodes, with branches, representing the list structures, extending from each of these tree nodes. As the tree is traversed within the optimizer, the lists, if present, will also have to be navigated before departing the node.

The optimizer component itself is generated by the Exodus tool known as the optimizer generator. The generator is a rule-based tool requiring an input of four elements to produce the necessary optimizer for a specific data model or application. As already stated, the operators are extended versions of the traditional relational algebra and the addition of the nest and unnest operators. Briefly, the four elements, placed in a description file include:

- Operators.
- Operator methods.
- Transformation rules.
- Implementation rules.

The operators are the actual operators required for the nested relational data model. These include at least the following operators:

- Project
- Select
- Join
- Nest
- Unnest

Other operators which will factor into the full-fledged model include the difference, the union, and the intersection operators. To provide more efficient operations within this particular data model, these operators will eventually need to be included during the development of the database system since they are necessary for set functionality.

In order to use operators, their implementations must be compiled and readily available as object modules within the system. The implementations may be numerous for each one of the operators. For example, the *project* operator may use either a file scan or a filtering implementation. Each implementation is known as an operator method and is consequently included in the description file of the optimizer generator.

The transformation and implementation rules deal primarily with how the operator and its methods are to be used. Depending upon the nature of the query, certain methods may or may not be possible. If the query is under optimization, these two rules in the description file are scanned to determine which methods are permissible, and of these permissible ones, which one is the most efficient. Again, the DBE determines what rules are required for these last two categories of the description file.

After the query tree undergoes a transformation according to the rules documented in the description file, the query is still in the form of a binary tree but the operators have been replaced by operator methods. The query is now closer to an actual implementation and another level of translation is required to obtain the actual source code for compilation. The query access plan, as it is now known, is passed to the query compiler.

Since the query optimization phase of the database system is extremely important, its implementation is as important as the nested relational data model itself. The goal of developing the data model is to seek increased performance in comparison to the traditional relational data model. Because of this aspect of the research, the optimizer rules are under design and implementation by another student and not within the scope of this project.

3.6 Query Compiler

The incoming query access plan (or *plan* for short) is now in an efficient structure ready for processing. The query compiler is ready to compile the generated E code into an object module. However, the plan, still in a tree-structure, must be transformed into E source code. That is, the tree must be "walked" to generate the E code.

The translation of the plan into E source code is performed by an important function referred to as the *tree-to-E* routine. The tree-to-E routine takes one node at a time and

transforms its data into an E source file. At each node, the operator method must be extracted and sent to the E file. Embedded within the plan are commands to form all intermediate relations or temporary relations if necessary, and the final relation inferred from the query. Intermediate relations may have to be established between upstream and downstream query nodes to store the extracted data. The tree-to-E routine must generate the structures for these intermediate and final relations.

The tree-to-E routine uses a depth-first traversal of the plan to basically climb the tree from bottom to top. This is necessary because the bottom nodes contain the references to the system's relation tables which all user queries must intially access. Between the other plan nodes, the data is sent downstream for further processing. In other words, the extracted information is sent to the next upper node in the plan tree. The development of each of the intermediate structures depends upon the auxiliary information associated with each of the plan nodes.

The auxiliary nodes attached to each of the query nodes support logical navigation throughout the nested relations. The nested relational data model requires a descent of some fashion into a top-level relation to the appropriate nested relation or relation-valued attribute for which the operator under consideration must apply its method. The auxiliary nodes maintain this navigation information to guide the operator to the queried nested relation as well as atomic attributes. Similar to walking the plan tree, the tree-to-E routine traverses the auxiliary node lists to translate this information into operations to be performed upon atomic and relation-valued attributes.

Eventually the tree-to-E routine reaches the top node of the plan tree. Once all the information in the auxiliary nodes are processed, the *main* routine must be appended to all other query declarations and definitions extracted from the plan tree. Because the query is the primary impetus of compiling all modules into a cohesive executable module, the addition of the *main* function occurs at this final stage of E code generation. The query is finally ready for transfer into the E compiler. The object modules and the generated query E code are gathered together for compilation together within the system's compiler.

3.7 E Compiler

The query is now located in what may be considered the "backend" of the database system. At this point, no more processing of the query will transpire since it is in its most efficient state. The tasks at hand focus on compiling the generated E code and then linking and loading it with the necessary object modules for query execution. The operator and access methods are implemented at this point and their object modules exist in an active mode for any number of compilations with user queries. That is, the query, after compilation, will be dynamically linked to the E run-time system (ERTS) (2). The relations within the database system are persistent, so the compilation must occur with the E compiler. Just linking the E library with the C compiler will not permit the handles to storage objects to be coupled with the executable module. All compilation, linking, and loading must occur employing the E compiler.

The target of the executable module is the Exodus storage manager. Although the nested relational database system is an executable module developed using the Exodus tools, the query is also an executable module requiring Exodus resources. The database program must manage the query in the form of a smaller program, directing the executable module to a file for driving the query. The system must set up a file to deposit the loaded code before execution of the query. Thus, a primary task of the "backend" is to handle system buffers and files for transferring of modules before final query execution.

3.8 Operator Methods

Since a query tree produced by the parser only contains the relational operators required for the user's query, methods must be maintained in the ERTS to implement the operations. However, each of the nested relational algebra operators may contain any number of methods to implement it. Two of the most popular methods perform some form of file scan of the relations in the storage manager, or a tuple filtering technique to create a stream of data leading from one or more relations into one final relation. Once the DBE designs and implements the operator methods, the object modules remain in the "backend" for all transient queries.

3.9 Access Methods

Along with operator methods, access methods such as B+ trees may enhance the efficiency of performing the query. Just as the operator methods, the access methods must be designed and implemented so that their objects will be available at the time of query compilation. For nested relations, access methods will still be operating on relations, although navigation methods embedded within the access plan exitting the query optimizer must guide the access method to the correct attribute. Just as with the operator methods, the E compiler links and loads the necessary access methods to the specific query.

3.10 Storage Manager

The storage manager is the component required to manage all storage objects instantiated within the database system. A number of functions are available to the DBE to handle specific objects. These functions allow the DBE to manipulate objects at the byte level, inserting and deleting bytes as deemed necessary. To enhance performance of a particular data model, storage object primitives may be used to direct database system operations at this level. However, to initially run the system at a higher-level of abstraction for the nested relational data model, this level is not tampered with since the storage manager adequately creates and modifies objects as required.

3.11 Database

The database works in conjunction with the storage manager to extract the necessary information from defined relations. The relations are instantiated objects permitting an arbitrary level of nesting to represent the nested relational data model.

3.12 Summary

The Exodus extensible architecture is comprised of tools and components required for designing a database system with respect to a specific application or data model. This research effort uses this development effort to construct a nested relational database management system.

IV. Design and Implementation

4.1 Nested Relation

A primary objective for implementing the nested relational data model is to increase system performance over other systems for non-traditional database applications. This performance translates into efficient retrieval of information with respect to other data models such as the traditional relational data model, the network model, and the hierarchical model. In addition to efficiency, the data model lends itself well to object-oriented design (OOD) (3), an increasingly popular methodology for transformation of real-world problems into the software domain. Data modeling of composite objects (10) also relies on the concept of nested relations.

Because of the relatively new approach of OOD and its corresponding languages (Ada, C++, Smalltalk), the design and implementation of nested relations has focused on other methodologies. Each of these may rely on such techniques as the physical storage of the data structures maintaining the data of a composite object, rather than the logical nature of the model. The logical structure of nested relations is not propagated throughout the entire system, a major goal of this thesis.

The Exodus database system's programming language allows for the placement of identically defined objects within a *collection*. In (2) and (5), dbclass's and dbstruct's are declared as collections of structures, where a structure represents the attributes of a relation. The collection is actually a relation, where the structures are tuples of the relation.

Instead of just placing single, atomic attributes within a structure, the collection construct is also allowed within a structure. In this manner, it is possible to implement the concept of nested relations. Exodus permits the data model to have a logical equivalent using this construct.

In addition to defining nested relations via collections, the logical nature of the data structure does propagate throughout all components comprising the system. Routines passing relation arguments can expect the same identical data structure throughout the entire system, from the catalog manager to the storage manager.

```
dbstruct child {
  dbchar name[20];
  dbint age;
  dbchar sex[2];
public:
  child(char *, int, char *);
  char * get_name();
  int get_age();
  char get_sex();
};
```

Figure 4.1. The *child* relation specification.

4.1.1 Implementation. The design of the nested relation calls for a simple, efficient, and logical method of translating the abstract concept of nested relations into the software domain. Exodus adds several constructs to the C++ programming language which permit such a definition. The *collection* construct and several associated commands guide the development of the nested relation.

The previously defined relations in Chapter 2 are used in the foregoing examples to illustrate implementation procedures for a nested relation. The *child* relation found in Figure 2.1, requires specifying its attributes using a structure construct. Since the relation is to be persistent, the structure must be prefixed with the two characters "db". Several other syntactic peculiarities distinguish Exodus structure definition from C++ and C structures and will be noted as necessary.

The structure definition for the *child* relation is shown in Figure 4.1. As part of the specification, private and public members exist to preserve data hiding features. The three private data elements are the attributes of the relation, while the member functions perform the insertion of data and access to each of the data elements. The constructor requires the three data types for the data elements when allocating space for a tuple. Since the member functions are public, they permit access to the relation data.

The semicolon appended to the dbstruct body is different than C, where none is required.

The implementation of the member functions are found in a ".e" file. In Figure 4.2, these functions basically use assignment of data from a dummy variable into the data

```
child::child(char * n, int a, char s) {
  char * dest1 = name;
  while(*dest1++ = n++);
  age = a;
  char * dest2 = sex;
  while(*dest2++ = *n++);
}

char * child::get_name() {
  dbchar * source = name;
  char * dest = new char[20];
  while(*dest++ = source++);
  return(&dest[0]);
}

int child::get_age() {
  return(age);
}

char child::get_sex() {
  char * source = sex;
  char * dest =  new char[2];
  while(*dest++ = *source++);
  return(&dest[0]);
}
```

Figure 4.2. Member function implementation for *child* relation.

member. A character-by-character transfer is used for string data. Exodus also permits the ease of implicitly converting "db" types to non-db ones during simple assignment. This allows the easy access and conversion of data for use in other routines. The double colon, "::", associates each object with its member functions.

Once the *child* relation is specified, it may be used within other structure definitions. To nest the *child* relation within the *employee* relation, a collection of *child* structures becomes the relation-valued attribute *children*. Figure 4.3 illustrates the *employee* relation.

The definition of the collection is a two-step process. First, a dbclass type must be specified for the collection of *child* tuples. Second, the *children* attribute is declared as the collection type *childRVA*. The "RVA" suffix is an abbreviation for "relation-valued

```
dbstruct employee {
  dbchar dept[5];
  dbchar name[20];
  dbint age;
  dbchar ssn[11];
  dbclass childRVA:collection[child];
  childRVA children;
public:
  employee(char *, char *, int, char *);
  char * get_dept();
  char * get_name();
  int get_age();
  char * get_ssn();
};
```

Figure 4.3. The nested relation specification for *employee*.

```
dbclass employee_rel:collection[employee];
persistent employee_rel employees;
```

Figure 4.4. Declaration of a persistent *employees* relation.

attribute."

The member functions refer to only the atomic attributes of the relation, while the collection of child structures are referenced by the C address-of operator, "&". By latching to the address of a particular *employee* tuple via the & operator, its *child* tuples may be further referenced, with all the member functions of the *child* object visible to the *employee* object. Thus, the recursive descent into nested relations is primarily possible by using the & operator.

The *employee* relation is actually a collection of these structures, so the letter "s" is appended to *employee* and a persistent object is declared as this collection of structures, shown in Figure 4.4.

Whenever the object is to be used in some application or query, an external variable declaration is used in the source file requiring the relation. The keywords "persistent" and "employeeRVA"(in this example) must also be included in the "extern" declaration.

4.2 The Parser

The parsing component drives the various functions of the database system. Several functions pertaining to the catalog manager provide input to the data dictionary, while a number of other procedures involve the structuring of the query tree. The parser passes the query tree to the query optimizer, while the query access plan is sent to the E source code translation routines via this component. In essence, this parser steers the functionality of the system.

Although Exodus expects a parser, no tool or editor is provided in developing the component. The UNIX tool, YACC, is a suggested instrument for constructing this element, as it was also used in the Exrel relational database system. YACC translates grammar rules in Backus-Nauer Form into the C programming language. Since E-specific language constructs cannot be used within a YACC file, the routines invoked within the parser must reside within an E file.

In modularizing the functionality of the parser, grammar rules separate the routines of the different components. Those procedures interacting with the catalog manager are grouped under data definition language statements, whereas query statements contain the query tree building routines. These latter statements guide the final query tree to the query optimizer, and pass the optimizer's output to the E source code routines. Upon execution of the query, the parser regains control of the system.

4.2.1 Implementation. The function of the parser is two-fold. First, it must interact with the catalog manager to insert data definitions into the data dictionary. Second, it controls the parsing of the query into a query tree representing the Colby relational algebra for nested relations. Since the catalog manager and the query tree structure rely on C++ and E constructs, the parser is function-driven, passing the necessary operands to their appropriate destinations.

Two separate data definition commands are possible when interacting with the catalog manager. To create the table types, **create type** prefixes the type information. In order to enter the *child* table types into the data dictionary, the following command is used for this purpose:

```
create type
child = (name:char(20), age:int, sex:char(2))
create type
employee = (dept:char(5), name:char(20), age:int,
ssn:char(11), children:child)
```

This information is placed in the data dictionary's symbol table as discussed below.

The second data definition command permits the declaration of actual nested relations for the database via the keywords **create table**. For example, to declare the *employees* relation, the command is used in the following manner:

```
create table
employees:employee.
```

As with the symbol table, the relation table maintaining the information for relations residing in the database is discussed below. The data for the relation must be present in a UNIX file, laid out in the following format:

```
[ top-level attr, top-level attr, ... { (nested attrs) ... }]
```

An example of this format, from the *employees* relation, may be illustrated in the following way:

```
[ Acct, Washington A.B., 33, 192-83-7465 { (Bob,5,M) (Carol,4,F)
}]
[ Eng, Carter G.H., 38, 325-96-0127 { (Kyle,10,M) (John,12,M)
(Lynne,6,M) }]
[ Dev, Kennedy E.F., 45, 519-73-3790 { (Mike,7,M) }]
[ Res, Lincoln C.D., 44, 234-61-9825 { (Tom,5,M) (Jeremy,4,M)
(Tiffany,7,F) (Anele,1,F) }]
```

After declaring the *employees* relation, the appropriate header files are constructed and implementation routines invoked to insert the data into the object.

The parser must assist in building the query tree structure for the query optimizer. Since there are three possible operators, the parser must recognize the following three query formats:

```
PJ { project list } ( relation or query )
SL ( relation or query ) [ condition ] { select list }
NJ ( relation or query ) { join path } ( relation or query )
```

These symbols represent the project, select, and natural join operators, respectively. Within parenthesis, another query may be nested instead of a relation, forming the sequence of relational operators for the query tree.

The parser is concerned with invoking the appropriate functions and passing the correct arguments. All three operators must construct an operator list, although the natural join operator list is known as a "join path." The parser passes the given relation-valued and atomic attributes with the *make_list*() function.

For each of the three operators, a query node must be created and the operator inserted into the correct portion of the node. The parser passes the operator to the query tree building routines through the function *make_qNode*(). If a relation name is part of the query, rather than a nested query, the name is also passed to the query node.

If a selection operator is undergoing parsing, the conditions or predicates also require separate predicate nodes. The square brackets delimiting a condition indicate that the enclosed arguments must be allocated space through the function call, *make_pred*(). This function passes information regarding attribute names, the operator to be used ($=, \neq, <$, ...), and the criteria to match the attribute against.

4.3 The Catalog Manager and Data Dictionary

An accurate account of all tables and their types must be maintained by the catalog manager component. The data elements comprising the data dictionary are placed into tables by the catalog manager. The use of C++ constructs allows easy insertion of data into the tables, as well as rapid access of the data.

name	*level*	*type*	*numb*	*parent*	*nest type*
child	SCHEME	ON_THE_FLY	3	SYSTEM	none
name	ATTR	CHAR	20	child	none
age	ATTR	INT	-1	child	none
sex	ATTR	CHAR	1	child	none
employee	SCHEME	ON_THE_FLY	5	SYSTEM	none
dept	ATTR	CHAR	5	employee	none
name	ATTR	CHAR	20	employee	none
age	ATTR	INT	-1	employee	none
ssn	ATTR	CHAR	11	employee	none
children	ATTR	RVA	3	employee	child

Figure 4.5. The Symbol Table.

Two data definition commands link the parser with the catalog manager. First, table types must be created before actual relations. The parser passes the attributes for a table type to the catalog manager, creating a symbol for the table type and its attributes. This includes relation-valued, as well as atomic attributes. Each symbol contains information to distinguish it from other symbols in the table, the attributes of that table representing the elements of the symbol table.

The second data definition command is used to insert relations into a second data dictionary table. Although a table type, as defined in the symbol table, refers to a particular relational structure, any number of actual objects may be declared as one of these table types. Once a table or relation is declared to be one of these table types, several routines are involved to construct a file to read in the data for the new relation. The relation data dictionary table must follow the progress of each created relation ensuring that its information coincides with that of the actual relation.

4.3.1 Implementation. The catalog manager receives data definition arguments from the parser's *create type* and *create table* commands. Two tables receive this data: the symbol table for table type information and the relation table for relations persisting in the database. From the *child* and *employee* table types and the *employees* relation, the symbol and relation tables appear as in Figure 4.5 and Figure 4.6.

relation name	*table type*
employees	employee

Figure 4.6. The Relation Table.

Since both tables are defined as collections, member functions are used by other components to access and obtain the necessary data.

4.4 The Query Tree

As an input object for the query optimizer, the query tree must be constructed to adhere to the optimizer's requirements. However, several parameters are flexible and permit the DBE to build the tree with a particular relational algebra in mind. The tree itself is nothing more than a series of query nodes linked together to logically access the database with respect to the relational operators.

Each query node represents a relational operator in the relational algebra, in this case the Colby algebra. One node may point to any number of other nodes, although this arity is normally set to two. This accounts for the fact that the standard relational operators are either unary (project, select) or binary (natural join). Besides the relational operator and the arity, the node must contain argument information. The argument allocates space for relation names, an operator list, and selection condition or predicate.

Since the argument provides the information for the relational operator for which the node was created, this requires additional design. Since the Colby algebra requires operator lists to recusively descend nested relations, the argument operator list must maintain this particular information.

In addition to the operator list branching out from the argument list structure, a selection condition may also form another branch. If a condition is applied to the top-level attributes of a relation, the argument selection data element points to the data structure maintaining this condition.

To construct the auxiliary branches for the operator lists and top-level conditions, two new actors must supplement the query tree structure. The operator list requires a list

operator	
argument	
input[0]	input[1]

Figure 4.7. The QUERY node.

name
pred
list

Figure 4.8. The ARGUMENT substructure.

node to maintain information of attributes to be projected or navigated across to reach other attributes. A predicate node is used to maintain information about attributes against which condition criteria are applied.

The query, as entered by the user and subsequently separated into query nodes by the parser, is formed in a top-to-bottom structure. The top node represents the first relational operator entered, while the bottom node requires information for the relation or relations involved within the query.

4.4.1 Implementation. The query tree requires three separate nodes for its construction. First, the query node is made of the elements found in Figure 4.7.

The operator is defined as a long integer, the two input elements as pointers to query nodes, and argument is a multiple element structure as defined in Figure 4.8.

"Name" is a character array for the relation that the relational operator is to operate on, "pred" is a pointer to a predicate, and "list" is a pointer to an operator list node.

The predicate node is structured as shown in Figure 4.9. The "op" element is a long integer, while the "left" and "right" elements are pointers to additional predicate nodes.

The list node, illustrated in Figure 4.10, maintains the information to navigate a nested relation. An "attribute descriptor" provides information for the attribute the node represents, the "condition" element points to a predicate, the "sublist" points to a nested

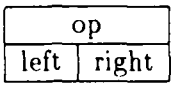

Figure 4.9. The PRED node.

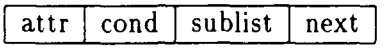

Figure 4.10. The LIST node.

relation, and the "next" element points to an attribute at the same level of nesting.

An example of a tree structure found in Figure 4.11, represents the following query pertaining to the *employees* relation:

```
PJ {name, children {name}} ( SL (employees) [age=33]
{children[age=5]}}
```

The resulting relation is shown in Figure 4.12.

Once the parser passes this structure to the query optimizer the optimizer transforms it into a query access plan.

4.5 The E Source Code Generator

The query tree structure will ultimately be transformed into a query access plan structure by the query optimizer. The plan represents the most efficient access into the database. The structure of the plan is similar to the query tree, with apparently minor changes to the primary node. Instead of an operator data element, the plan node defines an operator method data element. In the query tree, the relational operator is regarded in an abstract sense, while the plan's operator method represents the physical implementation of the operator. They include file scan, stream, and filtering methods, depending upon the operator. Other than the method element, plan nodes also require an argument element and pointers to additional plan nodes.

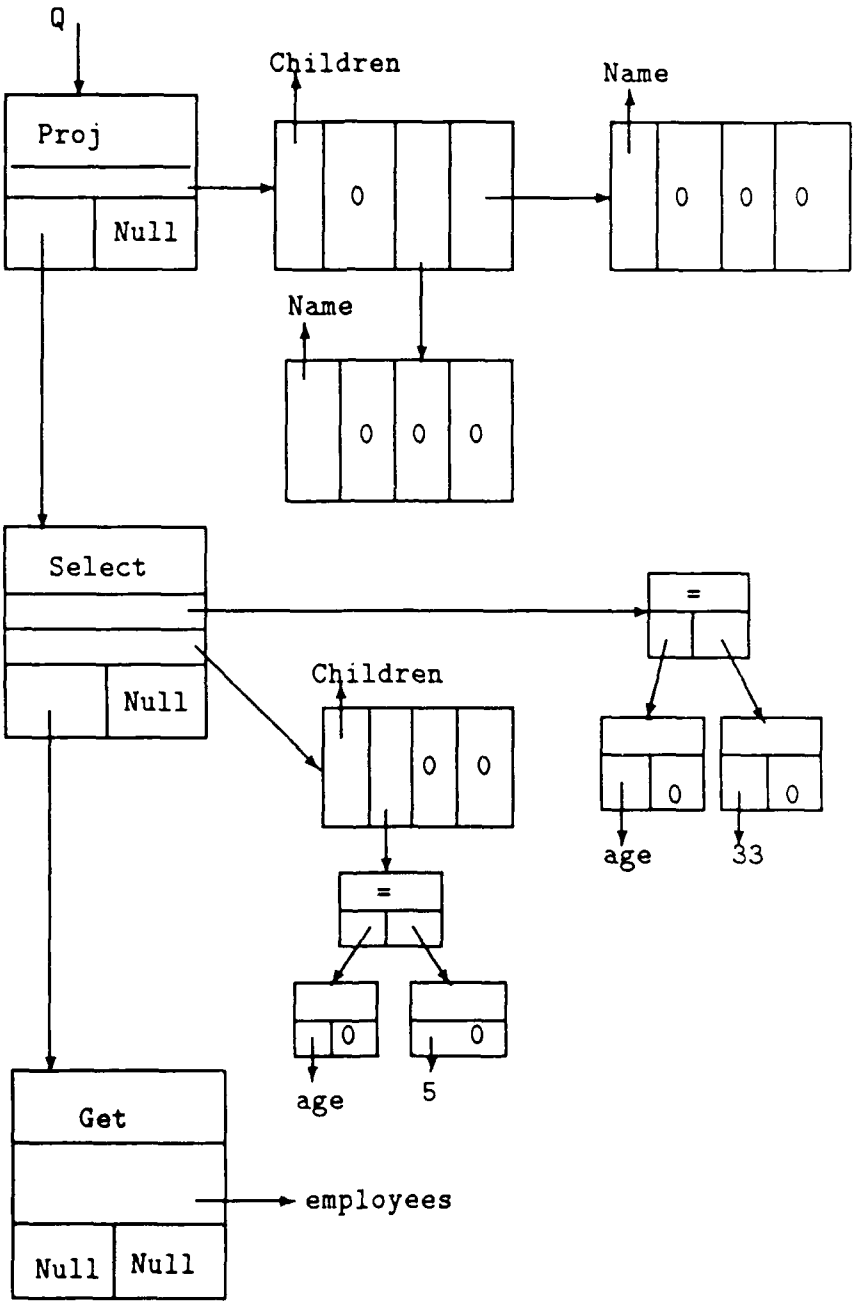

Figure 4.11. A query tree structure.

name	*children*
	name
Washington A.B.	Bob

Figure 4.12. The resulting relation.

operator method	
argument	
input[0]	input[1]

Figure 4.13. The PLAN node.

The plan nodes are to be examined one-by-one. The E source code generator relies on a "tree-to-e" routine which "walks" the plan and generates E source code according to the data in the plan nodes and their auxiliary branches. The tree-to-e routine initially descends to the bottom nodes, and works its way upward filling buffers with header file declarations, function and structure definitions, operator method class instantiations, and various external variable declarations for those relation or relations to be accessed. Once every plan node is traversed, a main routine is entered into a buffer. The buffers are placed together and named as an E file.

The E file is compiled and its object code is linked with the E run-time system (ERTS) (2). All object code for operator method implementations comprises the ERTS, and the query's object file is linked with the operators it requires. Once the query is completed in its execution, the memory allocated for the plan tree is released to the system for future use.

4.5.1 ***Implementation.*** The optimizer provides the query access plan, first to the optimizer, which hands it off to the code generating routines. The code generator expects a tree structure, much like the query tree structure, except that the plan node is structured as shown in Figure 4.13. Here, the integer representing a relational operator now represents an operator method.

The code generating routines require four separate buffers to place the appropriate

data and eliminate the possibility of early declarations with respect to other declarations. For instance, the query,

```
PJ { name, children {age} } (employees)
```

requires code to be generated using the file scan generator and a projection function. The E source code would first have to include the header file containing the *employees* relation specification. From the header file, the file scan can be declared for the tuples specified in the *child* and *employee* structures. The projection function would then be generated, along with the new data structure, which is reduced to the attributes of *name* and *age*. Finally, the main routine is generated with instantiations for the file scan. An iterator steps through each tuple and returns the data to the two attributes in the new relation. An iterator is applied to the new relation and the data is displayed to the screen. The generated code is shown in Figure 4.14 and Figure 4.15. A discussion of the file scan class and its implementation may be found in the following section.

4.6 The Operator Methods

The operator methods comprise the ERTS. After an operator is designed and implemented, its object code becomes a part of the ERTS, where these methods await linking with transient queries. Once the query is linked to the ERTS, the executable module invokes the operator methods and performs its member functions. The operator methods are objects which are formulated under an OOD methodology.

The operator methods, as generator classes, require certain parameters specific to the generator. Such parameters include relation and tuple types, where generator classes invoke iterator functions to process the relations and tuples. Since the methods are generator classes, the operators require a header file for the operator method specification and a file which contains the implementation code for the method. It is these classes which are instantiated within the E source code file generated via the query access plan.

The operator methods, as previously mentioned, may be one of several types. For instance, a file scan operator method may require a select or project function, depending

```
#ifndef FILE_SCAN_H
#include "file_scan.h"
#endif

#ifndef EMPLOYEES_H
#include "employees.h"
#endif

extern persistent employee_rel employees;

dbstruct temp1 {
  dbchar name[20];
public:
  temp1(char *);
  char * get_name();
};

temp1::temp1(char * n) {
  dbchar * dest = name;
  while(*dest++ = *n++);
}

char * temp1::get_name() {
  dbchar * source = name;
  char * dest = new char[20];
  while(*dest++ = *source++);
  return(&dest[0]);
}

dbstruct temp2 {
  dbchar name[20];
  dbclass temp1RVA:collection[temp1];
  temp1RVA temp1_rels;
public:
  temp2(char *);
  char * get_name();
};
```

Figure 4.14. The temporary relation structures.

```
temp2::temp2(char * n) {
  dbchar * dest = name;
  while(*dest++ = *n++);
}

char * temp1::get_name() {
  dbchar * source = name;
  char * dest = new char[20];
  while(*dest++ = *source++);
  return(&dest[0]);
}

dbstruct temp2_rel {
  dbclass temp2_relRVA:collection[temp2];
  temp2_relRVA temp2_rels;
};

dbclass temp2CT:collection[temp2_rel];
temp2CT Temp2;

class employee_rel_scan:file_scan[employee, temp2, employee_rel];

void sq1_proj (employee * e, temp2 * t2) {
  employee & e_ref  = * e;
  t2 = in (Temp2.temp2_rels) new temp2(e -> get_name);
  iterate(temp1 * t1 = e_ref.children.scan()) {
    t1 = in (e_ref.temp1_rels) new temp1(t1 -> get_name());
  }
}

main() {
  employee_rel_scan sq1(&employee_rel, NULL, sq1_proj);
  iterate(temp2 * t2 = sq1.next_tuple()) {
    cout << form("emp_name: %s\n", t2 -> get_name());
    temp2 & t2_ref = * t2;
    iterate(temp1 * t1 = t2_ref.temp1s.scan()) {
      cout << form("child_name: %s\n", t1 -> get_name());
    }
  }
}
```

Figure 4.15. Implementing the project query

upon the particular operator method residing in the plan node. Other forms of operator methods include streams and filters. A plan tree with a series of nodes normally send tuples upstream from one node to the next. The operator methods input these tuples and process the data according to their member functions. Figure 4.16 illustrates the specification of the file scan operator found in (2), while Figure 4.17 provides the loops join operator found in the same publication. In the file scan method, the selection function expects to have a destination type parameter since a supertuple may not place all of its elements into another tuple. This occurs because sets of data within a certain attribute may or may not meet the selection criteria.

4.7 Testing and Validation

After implementing the model within the Exodus architecture, several simple queries were applied to the database. The query tree structure was built for most combinations of the project, select, and join operators. The E code generators produced code for the query for one level of nesting, invoking either the file scan or loops join operators. No performance tests have been conducted on the system.

```
class file_scan [
  dbclass srcType {},
  dbclass dstType {},
  dbclass srcRelType { public: iterate srcType * scan(); }
]
{
  typedef int  (*selFunc)(srcType*, dstType*);
  typedef void (*projFunc)(srcType*, dstType*);
  srcRelType * relation;
  selFunc      select;
  projFunc     project;
public:
  file_scan(srcRelType*, selFunc, projFunc);
  iterator dstType * next_tuple();
};

file_scan::file_scan(srcRelType * rptr, selFunc sPtr, projFunc
pPtr) {
  relation = rPtr;
  select   = sPtr;
  project  = pPtr;
}

iterator dstType * file_scan::next_tuple() {
  dstType rsltTuple;
  iterate(srcType * tuplePtr = relation -> scan()) {
    if ((select == NULL) || (select(tuplePtr))
      if (project != NULL) {
        project(tuplePtr, &rsltTuple);
        yield(&rsltTuple);
      }
      else
        yield ((dstType*)tuplePtr);
}
```

Figure 4.16. File scan class and implementation.

```
class loops_join [
  dbclass srcType1 {},
  dbclass srcType2 {},
  dbclass dstType {},
  class subQuery (public: iterator srcType1 * next_tuple(); },
  dbclass innerRelType { public: iteator srcType2 * scan(); }
]
{
  typedef int (*matchFunc)(srcType1*, srcType2*);
  typedef void (*joinFunc)(srcType1*, srcType2*, dstType*);
  subQuery *     outer;
  innerReltype * inner;
  matchFunc      match;
  joinFunc       join;
public:
  loops_join(subQuery*, innerRelType*, matchFunc, joinFunc);
  iterator dstType * next_tuple();
};

loops_join::loops_join(subQuery * query, innerRelType * innerRel,
matchFunc matchPtr, joinFunc joinPtr) {
  outer = query;
  inner = innerRel;
  match = matchPtr;
  join = joinPtr;
}

iterator dstType * loops_join::next_tuple() {
  dstType rsltTuple;
  iterate(srcType1 * outerTuple = outer -> next_tuple())
    iterate(srcType2 * innerTuple = inner -> scan())
      if ((match == NULL) || match(outerTuple, innerTuple)) {
        join(outerTuple, innerTuple, &rsltTuple);
        yield(&rsltTuple);
      }
}
```

Figure 4.17. Loops join class and implementation.

V. Conclusion

5.1 Summary

This thesis effort accomplished several objectives, resulting in the design and implementation of a nested relational database system under the Exodus extensible database architecture. The key objectives include:

- Nesting relations within relations.
- A parser to create and maintain a persistent data dictionary.
- Implementing the Colby relational algebra.
- Implementing operator methods for selection, projection, and natural join of nested relations.

An initial objective of this effort concerned the implementation aspects of nested relations. Through the use of the collection construct provide by Exodus, structures which defined attributes of a particular table type were permitted to represent single entities or attributes within other table types. By mapping corresponding relations to collections, the logical nature of nested relations used standard procedures thoughout all the system components. Also, the nested relations were easy to navigate because of their logical structure.

A parsing component was designed and implemented to permit the creation of nested relations. The data dictionary required table type information to be separated into its individual attributes and placed into the symbol table for future reference. Since nested relations had relations as their attributes, the data dictionary allowed previously defined table types to become table types within other relations against which their relation-valued attributes were declared. The persistence feature of Exodus allowed the data dictionary to remain in the storage manager between program executions.

As an offshoot of parsing data for the data dictionary, a format to enter actual data from a UNIX file into a relation allowed a rapid and simple method of loading it into the relation. The UNIX file followed a logical layout for the data file in conjunction with the

relation it was to fill. The creation of such a format, logically delimited by braces, brackets, parenthesis, and commas, provided an easy vehicle for loading data into nested relations.

The relational algebra proposed by Colby was parsed and inserted into the operator data elements of the query tree. The query nodes and the tree structure resembled a general structure required by the query optimizer. Because of the flexible nature of the query tree and the nodes which described it, the query tree represented the Colby relational algebra and was, thereby, suitable for manipulating nested relations.

The methods to implement the project, select, and natural join operators resided as object files in the ERTS. Here, query code was compiled and linked with the implemented methods to execute the query. The operator methods used functions to recursively descend relations, applying criteria to carry out the necessary functions.

5.2 Future Recommendations

Several enhancements may be added to the current system to improve future useability. First of all, the parser currently recognizes the three primary relational operators, project, select, and natural join. Additional Colby operators may be added to the system, including nest, unnest, and the set operations. To implement these operators, additional methods may be added to the ERTS along with access methods. Finally, a more user friendly query language, such as SQL/NF, may be added to the front end of the database system.

Vita

Captain Michael A. Mankus [redacted] He attended Purdue University at West Lafayette, Indiana for one year before entering the United States Air Force Academy at Colorado Springs, Colorado. There he received his Bachelor of Science degree in Electrical Engineering as well as his Commission as a Second Lieutenant in the United States Air Force in June 1985. His first duty assignment was as a Command and Control Test and Evaluation Engineer at the 1815th Operational Test and Evaluation Squadron, Wright-Patterson Air Force Base, Dayton, Ohio. He entered the School of Engineering, Air Force Institute of Technology in June 1988 and graduated December 1989. His current address is at Nellis Air Force Base, Nevada.

[redacted]

Bibliography

1. "Ingres/SQL Reference Manual". (August, 1986).
2. "An Overview of the Exrel Relational DBMS". pages 1–16, (May, 1989.
3. Banerjee, J and others. "Data Model Issues for Object-Oriented Applications". *ACM Transactions on Office Information Systems*, 5(1):3–26, (January 1987).
4. Carey, M.J., Dewitt, D.J., Daniel, F., Graefe, Goetz, Richardson, J.E., Shekita, E.J., and Muralikrishna, M. "*The Architecture of the EXODUS Extensible DBMS*". Morgan Kaufmann Publishers, San Mateo, CA, 1988.
5. Carey, M.J., DeWitt, D.J., Richardson, J.E., and Shekita, E.J. "Object and File Management in the EXODUS Extensible Database System". *Proceedings of the 12th VLDB Conference*, (August 1986).
6. Codd, E.F. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM*, 13(6), (June 1970).
7. Colby, Latha S. "A Recursive Algebra for Nested Relations". Technical report, Indiana University, (January 1989).
8. Date, C.J. *An Introduction to Database Systems Volume II*. Addison-Wesley, Reading, Massachusetts, 1983.
9. Hafez, Aladdin and Ozsoyoglu, Gultekin. "Storage Structures for Nested Relations". *Data Engineering*, pages 31–38, (September 1988).
10. Kim, Won, Chou, Wong-Tai, and Banerjee, Jay. "Operations and Implementation of Complex Objects". *IEEE Transactions on Software Engineering*, 14(7):985–995, (July 1988).
11. Korth, H.F. and Silberschatz, A. *Database System Concepts*. McGraw-Hill Book Company, New York, N.Y., 1986.
12. Roth, M. A. and others. "SQL/NF: A Query Language for ¬1NF Relational Databases". *Information Systems*, 12(1):99–114, (January 1987).
13. Roth, M.A., Korth, H.F., and Silberschatz, A. "Extended Algebra and Calculus for Nested Relational Databases". *ACM Transactions on Database Systems*, 13(4):389–417, (December 1988).
14. Valduriez, Patrick, Khoshafian, Setrag, and Copeland, George. "Implementation Techniques of Complex Objects". *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 101–109, (August 1986).

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89D-11	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology(AU) Wright-Patterson AFB, Ohio 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
DESIGN AND IMPLEMENTATION OF THE NESTED RELATIONAL DATA MODEL UNDER THE EXODUS EXTENSIBLE DATABASE SYSTEM

12. PERSONAL AUTHOR(S)
Michael A. Mankus, B.S.E.E., Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1989 December	63

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	05		Mathematical and Computer Sciences; Computer Programming and Software

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Major Mark A. Roth, USAF
Associate Professor of Computer Systems

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Maj Mark A. Roth, Associate Professor	22b. TELEPHONE (Include Area Code) (513)255-3576
	22c. OFFICE SYMBOL ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

The problem addressed in this thesis effort concerns the design and implementation of the nested relational data model. The data model is designed within the Exodus extensible architecture. Although a large amount of theory exists with the model, no vehicle has been available to implement the concepts. The objective of the model is to increase performance of non-traditional databases by modeling real-world objects in the problem domain into nested relations within the software domain of Exodus.

Exodus is used to implement several components essential to the data model. First, the concept of nested relations is realized, and then a parser is developed to create and maintain a data dictionary. The Colby relational algebra is used to form the query tree for the query optimizer, and a plan tree permits the code to be generated for the query. Operator methods are developed for the query to be subsequently executed.

The nested relational data model was implemented using the Exodus architecture. The query tree was built and the code generated for the architecture's compiler. Operator methods were implemented for the project, select, and natural join operators. Because the data model can be implemented, more non-traditional databases can be developed with efficient components.